ARbitrary PRECision Computation Package (ARPREC)

==========

Revision date: 25 Oct 2012

Authors:
David H. Bailey Lawrence Berkeley Natl Lab
dhbailey@lbl.gov
Yozo Hida U.C. Berkeley
yozo@cs.berkeley.edu
Karthik Jeyabalan now at Cornell University
kj44@cornell.edu
Xiaoye S. Li Lawrence Berekely Natl Lab
xiaoye@nersc.gov
Brandon Thompson now at Synopsis
thompsbj@gmail.com

C++ usage guide:
Alex Kaiser Lawrence Berkeley Natl Lab
adkaiser@lbl.gov

This work was supported by the Director, Office of Science, Division of Mathematical,
Information, and Computational Sciences of the U.S. Department of Energy under contract
numbers DE-AC03-76SF00098 and DE-AC02-05CH11231.

*** IMPORTANT NOTES:

See the file COPYING for modified BSD license information.
See the file INSTALL for installation instructions.
See the file NEWS for recent revisions.
See the mostly identical file README.pdf for additional tables on selected functions.

Outline:

I. Introduction

A. Overview

ARPREC is a software package for performing arbitrary precision

arithmetic. It consists of a
revision and extension of Bailey's earlier MPFUN package, enhanced with special IEEE
numerical techniques. Features include:

1. Written in C++ for broad portability and fast execution.

2. Includes C++ and Fortran 90/95 interfaces based on custom datatypes and
operator/function overloading, which permit the library to be used with only minor
modifications for many conventional C++ and Fortran-90 programs.

3. Includes all of the usual arithmetic operations, as well as many transcendental functions,
including cos, sin, tan, arccos, arcsin, arctan, exp, log, log10, erf, gamma and bessel functions.

4. Supports three arbitrary precision datatypes: mp_real, mp_int and mp_complex.

5. Supports many mixed-mode operations between arbitrary precision variables or
constants and conventional variables or constants.

6. Includes special library routines, incorporating advanced algorithms for extra-high
precision (above 1000 digits) computation.

7. Includes a number of sample application programs, including programs for quadrature
(numerical definite integrals), PLSQ (integer relation finding) and polynomial root finding.

8. Includes the "Experimental Mathematician's Toolkit". This is a
   self-contained interactive program that performs many operations
   typical of modern experimental mathematics, including
   arithmetic expressions, common transcendental functions,
   infinite series evaluation, definite integrals, polynomial roots,
   user-defined functions, all evaluated to a user-defined level of
   numeric precision, up to 1000 decimal digits.

An overview of C++ and Fortran usage is given below. A detailed description of the algorithms used is available in the doc subdirectory (see doc/arprec.ps).

B. Directory Structure

There are eight directories and several files in the top level directory,

which are described below:

config This directory contains auxiliary files used by the configure script.
doc This directory contains a paper describing the algorithms.
fortran This directory contains Fortran-90 module files.
include This directory contains the header files for C++ codes.
src This contains the source code of the ARPREC library. This source code does not include inline functions, which are found in the header files in the include/ directory.
The archive library file is placed in this directory when
the make operation is performed.
tests This directory contains test programs to sanity check the
the compiled library (through "make check"). It also include
a number of sample C++ programs, including high precision
quadrature and PSLQ (integer relation detection).
toolkit This directory contains the Fortran-90 code for the Toolkit.
A C++ version of the Toolkit is under development.

II. C++ usage

Please note that all commands refer to a Unix-type environment such as Mac OSX or Ubuntu
Linux using the bash shell.

A. Building

To install the ARPREC system, the g++ and, if Fortran is to be used, an appropriate
Fortran compiler must be installed. See INSTALL for instructions for your particular system.

To build the library, first run the included configure script by typing

```
./configure
```

This script automatically generates makefiles for building the library and selects compilers
and necessary flags and libraries to include. If the user wishes to specify compilers or flags

they may use the following options.

CXX	C++ compiler to use
CXXFLAGS	C++ compiler flags to use
CC	C compiler to use (for C demo program)
CFLAGS	C compiler flags to use (for C demo program)
FC	Fortran 90 compiler
FCFLAGS	Fortran 90 compiler flags to use
FCLIBS	Fortran 90 libraries needed to link with C++ code.

For example, if one is using GNU compilers, configure with:

```
./configure CXX=g++ FC=gfortran
```

The Fortran and C++ compilers must produce compatible binaries. On some systems
additional flags must be included to ensure that portions of the library are not built with 32 and 64 bit object files. For example, on 64-Bit Mac OSX 10.6 (Snow Leopard), OSX 10.7 (Lion) and OSX 10.8 (Mountain
Lion), the correct configure line using the gfortran compiler is:

```
./configure CXX=g++ FC=gfortran FCFLAGS=-m64
```

To build the library, simply type

```
make
```

and the automatically generated makefiles will build the library including archive files.

To allow for easy linking to the library, the user may also wish to install the archive files to a standard place. To do this type:

```
make install
```

This will also build the library if it has not already been built. Many systems, including Mac and Ubuntu Linux systems, require administrator privileges to install the library at such standard places. On such systems, one may type:

```
sudo make install
```

instead if one has sufficient access.

The directory "tests" contains programs for high precision quadrature and integer-relation detection. To build such programs, type:

```
make demo
```

in the "tests" directory.

B. Linking

The simplest way to link to the library is to install it to a standard place as described above,
and use the Đl option. For example

```
g++ compileExample.cpp -o compileExample -l arprec
```

One can also use this method to build with make. A file called ÒcompileExample.cppÓ and the
associated makefile ÒmakeCompileExampleÓ illustrate the process.

A third alternative is to use a link script. If one types "make quads" in the test directory, the
output produced gives guidance as to how to build the files. By following the structure of
the compiling commands one may copy the appropriate portions, perhaps replacing the
filename with an argument that the user can include at link time. An example of such a
script, which also links to included quadrature routines, is as follows:

```
g++ -DHAVE_CONFIG_H   -I.. -I../include -I../include   -O2 -Wall -MT
$1.o -MD -MP -MF
.deps/$1.Tpo -c -o $1.o $1.cpp

mv -f .deps/$1.Tpo .deps/$1.Po

g++  -O2 -Wall   -o $1 $1.o util.o arprec-integrate.o quad-erf.o quad-
gs.o quad-ts.o
../src/libarprec.a
```

To use it, make the link script executable and type:

```
./link.scr compileExample
```

Note that the file extension is not included because the script handles all extensions,
expecting the source file to have the extension Ô.cppÕ .

C. Programming techniques

As much as possible, operator overloading is included to make basic programming as much
like using standard typed floating-point arithmetic. Changing many codes should be as

simple as changing type statements and a few other lines.

i. Constructors

To create mp_real variables calculated to the proper precision, one must use care to use the
included constructors properly. Many computations in which variables are not explicitly
typed to multiple-precision may be evaluated with double-precision arithmetic. The user
must take care to ensure that this does not cause errors. In particular, an expression such as
1.0/3.0 will be evaluated to double precision before assignment or further arithmetic. Upon
assignment to a multi-precision variable, the value will be zero padded. This problem is
serious and potentially difficult to debug. To avoid this, use the included constructors to
force arithmetic to be performed in the full precision requested. For example:

```
mp_real x = mp_real("1.0") / 3.0 ;
```

or

```
mp_real x = mp_real("1.0") ;
x /= 3.0 ;
```

or

```
mp_real x  = Ò1.0Ó  ;
x /= 3.0 ;
```

ii. Included functions and Constants

Supported functions include assignment operators, comparisons, arithmetic and
assignment operators, and increments for integer types. Standard C math functions such as
exponentiation, trigonometric, logarithmic and exponential functions are included.
Additional functions for gamma, erf, rounding, etc. are also implemented. As in assignment
statements, one must be careful with implied typing of constants when using these
functions. Many codes need particular conversion for the power function, which is
frequently used with constants that must be explicitly typed for multi-precision codes.

Constants included, which are global and calculated to full precision upon the call of
mp_init, with type signatures, are:

```
static mp_real _pi;
static mp_real _log2;
static mp_real _log10;
static mp_real _eps;
```

where Ò_epsÓ is the accuracy of the current state as initialized by the user.

ii. Conversion of types

Static casts may be used to convert constants between types. One may also use constructors
to return temporary multi-precision types within expressions, but should be careful, as this
will waste memory if done repeatedly. For example:

```
mp_real y ;
y = gamma( mp_real(4.0) / 3.0 ) ;
```

CÐstyle casts may be used, but are not recommended. Dynamic and reinterpret casts are
not supported and should be considered unreliable. Casting between multi-precision and
standard precision types can be dangerous, and care must be taken to ensure that programs
are working properly and accuracy has not degraded by a misplaced type-conversion.

D. Control and Modification of Precision Levels

There are a number of functions for controlling precision levels. One must call mp_init
before performing any multi-precision computations or initializing any constants. The
relevant functions are described in more detail in the table in the corresponding README.pdf

E. I/O

The standard I/O stream routines have been overloaded to be fully compatible with all
included data types. One may need to manually reset the precision of the stream to obtain
full output. For example, if 500 digits are desired, type

```
cout.precision(500) ;
```

When reading values using cin, each input numerical value must start on a separate
line, and end with a comma. Three formats are acceptable:

1. Write the full constant
2. 10 ^ exponent x mantissa
3. mansissa e exponent

Here are four valid examples:

```
1.1,
3.14159 26535 89793,
10^3 x 3.14159 26535 89793 23846
        26433 83279 50288,
123.123123e5000
```

When read using cin, these constants will be converted using full multi-precision accuracy.

III. Fortran-90 Usage

A. Fortran-C++ interoperability

The configure script attempts to guess what flags and libraries to use, but it is not perfect on all platforms. There are differing conventions among computer vendors as to how to handle a C function called from a Fortran program. For example, Sun's f90 compiler attaches an underscore to all function names, while Cray's cf90 compiler makes the name all uppercase. IBM's compiler leaves these names alone. The configure script should detect the correct convention in most cases. If it doesn't, the appropriate transformation can be defined in the config.h file. If you encounter difficulties, please notify us so that the configure script can handle the situation more gracefully.

To link a Fortran-90 program with the C++ arprec library, you must link with the C++ compiler used to generate the library, as this will insure that data in the C++ library routines will be properly instantiated. The Fortran 90 interface is found in arprecmod library. The C++ main entry is found in arprec_f_main library. The arprec-config script installed during "make install" can be used to determine which flags to pass to compile and link your programs:

```
  arprec-config --fcflags
      displays compiler flags needed to compile your Fortran files.
  arprec-config --fclibs
```

```
      displays linker flags needed by the C++ linker to link in all
the
      necessary libraries.
  arprec-config --fmainlib
      display linker flags needed if your main program is written
      in Fortran.  You must rename your main Fortran program as
      "subroutine f_main".
```

A sample Makefile that can be used as a template for compiling Fortran programs using ARPREC library is found in fortran/Makefile.sample.

An alternate way of handling all of these flags is simply to type, for instance, "make quadts". This compiles the Fortran program tquadts.f, links with all necessary library files, and produces the executable "quadts". As this is being done, all flags and linked libraries are displayed. For instance, on many Mac systems, presuming g++-4.0 was defined for C++ and gfortran for F90, the following is output:

```
  gfortran  -O2 -ffree-form -c -o tquadts.o tquadts.f
  g++-4.0  -O2 -Wall   -o quadts tquadts.o second.o libarprec_f_main.a
  libarprecmod.a ../src/libarprec.a
  -L/usr/local/lib/gcc/i386-apple-darwin9.0.0/4.3.0
  -L/usr/local/lib/gcc/i386-apple-darwin9.0.0/4.3.0/../../.. -
lgfortranbegin
  -lgfortran
```

Thus a general compile-link script (which could be saved in an executable file named "complink.scr") is the following:

```
  gfortran  -O2 -ffree-form -c -o $1.o $1.f
  g++-4.0  -O2 -Wall   -o $1 $1.o second.o libarprec_f_main.a \
  libarprecmod.a ../src/libarprec.a \
  -L/usr/local/lib/gcc/i386-apple-darwin9.0.0/4.3.0 \
  -L/usr/local/lib/gcc/i386-apple-darwin9.0.0/4.3.0/../../.. -
lgfortranbegin \
  -lgfortran
```

Note that if the .f90 suffix is used for Fortran-90 source files, the -ffree-form flag can be omitted, and the first line above ends with "$1.f90". Remember to type "chmod +x complink.scr". Then, for instance, a program named "prog.f90" could be compiled and linked by merely typing "./complink.scr prog".

B. Fortran programming techniques

The basic concept of the ARPREC F90 package is to extend the Fortran-90 language, by means of completely standard Fortran-90 features (operator overloading and custom datatypes), to perform

computations with an arbitrarily high level of numeric precision (hundreds or thousands of digits). In most cases, only minor modifications need to be made to an existing Fortran program -- e.g., change the type statements of variables that you wish to be treated as multiprecision variables, plus a few other minor details. Three multiprecision datatypes are supported: mp_real, mp_int and mp_complex. The ARPREC F90 package is based on the earlier MPFUN90 package, which is available in the same online directory as ARPREC. Recently the MPFUN90 package was revamped to be completely compatible, at the Fortran-90 user level, with the ARPREC F90 package.

Several example application programs using the F90 arprec software can be found in the fortran and the toolkit directory.

To use the package, first set the global variable mpipl, which is the maximum precision level in digits, in the file mp_mod.f. As delivered, mpipl is set to 2000 digits, which means that the package can handle user programs specifying up to 2000 digit accuracy. If mpipl is changed, mp_mod.f must be recompiled. With mpipl set to 2000 digits, programs can define a working precision level to up and including 2000 digits.

Modifying an existing Fortran-90 program to use the ARPEC library is generally quite easy, because of the translation facilities in mp_mod.f. A sample user program is:

```
  subroutine f_main
    use mpmodule
    implicit none
    type (mp_real) a, b

    call mpinit (500)
    a = 1.d0
    b = cos(a)**2 + sin(a)**2 - 1.d0
    call mpwrite(6, b)
    stop
  end program
```

This verifies that cos^2(1) + sin^2(1) = 1 to 500 digit accuracy. The user's main program must be a subroutine named "f_main". This is required so that certain C++ initialization is performed. The line "use mpmodule", as shown above, must be included at the beginning of each subroutine or function subprogram that uses multiprecision datatypes. Multiprecision variables are declared using a Fortran-90 defined type statement such as the following.

```
   type (mp_real) a, b, c(10)
   type (mp_integer) k1, k2, k3
   type (mp_complex) z1, z2(5,5), z3
```

Most operators and generic function references, including many mixed-mode type combinations, have been overloaded (extended) to work with multiprecision data. It is important, however, that users keep in mind the fact that expressions are evaluated strictly according to conventional Fortran operator precedence rules. Thus some subexpressions may be evaluated only to real*4 or real*8 accuracy. For example, with the code

```
real*8 d1
type (mp_real) t1, t2
...
t1 = cos (t2) + d1/3.d0
```

the expression d1/3.d0 is computed to real*8 accuracy only (about 15 digits), since both d1 and 3.d0 have type real*8. This result is then converted to mp_real by zero extension before being added to cos(t2). So, for example, if d1 held the value 1.d0, then the quotient d1/3.d0 would only be accurate to 15 digits. If a fullly accurate multiprecision quotient is required, this should be written:

```
real*8 d1
type (mp_real) t1, t2
 ...
t1 = cos (t2) + mpreal (d1) / 3.d0
```

which forces all operations to be performed with multiprecision arithmetic.

Along this line, a constant such as 1.1 appearing in an expression is evaluated only to real*4 accuracy, and a constant such as 1.1d0 is evaluated only to real*8 accuracy (this is according to standard Fortran conventions). If full multiprecision accuracy is required, one should write

```
type (mp_real) t1
...
t1 = '1.1'
```

The quotes enclosing 1.1 specify to the compiler that the constant is to be converted to binary using multiprecision arithmetic, before assignment to t1. Quoted constants may only appear in assignment statements such as this.

The ARPREC F90 package does NOT support mixed-mode operations between "real" (single precision real, ie real*4) data and multiprecision data, nor between "complex" (single precision real, ie complex*8) data and multiprecision data. If your programs have such datatypes, convert these to real*8 and complex*16, respectively, before attempting to change the program to use ARPREC.

C. Functions defined with multiprecision arguments

F90 functions defined with mp_int arguments:
  Arithmetic: + - * / **
  Comparison tests: == < > <= >= /=
  Others: abs, max, min

F90 functions defined with mp_real arguments:
  Arithmetic: + - * / **
  Comparison tests: == < > <= >= /=
  Others: abs, acos, aint, anint, asin, atan, atan2, cos, dble, erf,
  erfc, exp, int, log, log10, max, min, mod, mpcsshr (cosh and sinh),
  mpcssnf (cos and sin), mpranf (random number generator in (0,1)),
  mpnrtf (n-th root), sign, sin, sqr, sqrt, tan

D. Input/output of multiprecision data

Input and output of multiprecision data is performed using the special subroutines mpread and mpwrite. The first argument of these subroutines is the Fortran I/O unit number, while additional arguments (up to 9 arguments) are scalar variables or array elements of the appropriate type. Example:

```
type (mp_real) a, b, c(n)
...
call mpread (6, a, b)
do j = 1, n
  call mpwrite (6, c(j))
enddo
```

When using mpread, each input numerical value must start on a separate line, and end with a comma. Here are three valid examples:

```
1.1,
3.14159 26535 89793,
10^3 x 3.14159 26535 89793 23846
  26433 83279 50288,
```

When read using mpread, these constants will be converted using full multiprecision accuracy.

One can also read and write multiprecision variables and arrays using Fortran unformatted (binary) I/O, as in

```
type (mp_real) t1, a(30)
write (11, t1)
write (12, a)
```

Data written to a file in this fashion can be read with a similar unformatted read statement, but only on the same system that it was

written on. Unformatted files written by MPFUN90 programs are not compatible with unformatted files written by ARPREC.

E. Handling precision level

The initial working precision, and the maximum for this run, is set by

```
call mpinit (idig)
```

where idig is the desired precision in digits. The current working precision, in digits (idig) and words (iwds), can be obtained by

```
call mpgetprec (idig)     [or]
call mpgetprecwords (iwds)
```

(one word contains 48 bits, or about 14.44 digits). The working precision level may be changed by calling

```
call mpsetprec (idig)     [or]
call mpsetprecwords (iwds)
```

The output precision, which by default is limited to 56 digits no matter what the working precision level, is given in the global variable mpoud, and may be changed or fetched simply by writing

```
mpoud = idig     [or]
idig = mpoud
```

This may optionally be done by using subroutine calls:

```
call mpsetoutputprec (idig)    [or]
call mpgetoutputprec (idig)
```

F. Multiprecision system variables

A number of global variables are defined in the Fortran-90 wrapper, which may be accessed and in some cases changed by the user. Here is a listing and brief description of these variables:

Integer parameters [cannot be changed by the user during execution]:

mpipl	Max prec level, digits	User sets in mp_mod.f
mpldb	Logical unit for some output (=6)	Set in mp_mod.f
mpnbt	Bits per word (= 48)	Set in mp_mod.f
mpwds	Max prec level in words	Set in mp_mod.f, from mpipl

Integer variables [may be changed by user during exeuction]:

mpoud	Number of digits output with mpwrite; default = 56

Integer variables [accessed via mpgetpar/mpsetpar (see note below)]:

```
mpidb      Debug level for arprec routines; default = 0
mpier      Error flag and error number; default = 0
mpird      Rounding mode (0, 1 or 2); default = 1
mpmcr      Threshold for advanced routines; default = 7
mpndb      Number of debug words output; default = 22
mpker      Array of 72 error handling flags; default each entry = 2
```

Multiprecision real parameters [calculated during initialization]:

```
mpl02      Log(2)
mpl10      Log(10)
mppic      Pi
mplrg      Very large mp_real value = 2^(48*2^27)
mpsml      Very small mp_real value = 2^(-48*2^27)
```

Multiprecision real variables [may be changed by user during exeuction]:

```
mpeps      Epsilon for user program; default = 10^(-prec)
```

The system parameters mpidb, mpier, mpird, mpmcr, mpndb and mpker can be stored or fetched as follows:

```
   integer mpidb
   ...
   call mpsetpar ('mpndb', 10)         [or]
   call mpgetpar ('mpidb', mpidbx)
```

IV. The Experimental Mathematician's Toolkit

A. Installation instructions:

1. Download and install the ARPREC library (see instructions in README of the main arprec directory). Then type "make toolkit" and enter the toolkit directory.

2. Run the mathinit program by typing ./mathinit. This computes a few constants and then generates quadrature data for tanh-sinh quadrature. This takes about 2 minutes to run, depending on the system, and generates two files, one about 19 Mbyte long. By default, only data for the tanh-sinh quadrature scheme is currently computed. To compute data for all three quadrature schemes, see the comments at the
start of mathinit.f, or below in Section C, Item 13.

3. The mathtool program may now be invoked by typing ./mathtool.

B. Usage instructions:

Here are some examples, as shown in the prompt of the mathtool program:

```
e + pi + log2 + log10 + catalan  Adds these pre-defined constants.
result[1] + result[2]            Adds result #1 to result #2.
alpha = arctan[3] - 3*log[2]     Defines or sets user variable alpha.
fun1[x,y] = 2*sqrt[x]*erf[y]     Defines user function fun1.
clear[nam1]                      Clears definition of variable or
function.
integrate[1/(1+x^2), {x, 0, 1}]  Integrates 1/(1+x^2) from x=0 to 1.
sum[1/2^k, {k, 0, infinity}]     Sums 1/2^k from k=0 to infinity.
binomial[20,10]*factorial[10]    Evaluates binomial coeff and
factorial.
zeta[3] + zetaz[1,1,2]           Evaluates zeta and multi-zeta
functions.
table[x^k, {k, 1, 4}]            Forms the list [x^1, x^2, x^3, x^4].
pslq[table[x^k, {k, 0, n}]]      Finds coeffs of degree-n poly for x.
polyroot[1,-1,-1,{0.618}]        Finds real root of 1-x-x^2=0 near
0.618.
polyroot[1,2,3,{-0.33, 0.47}]    Complex root of 1+2x+3x^2 near
-0.33+0.47i.
digits = 200                     Sets working precision to 200 digits.
epsilon = -190                   Sets epsilon level to 10^(-190).
eformat[190,180]                 Display using E format with 180
digits.
fformat[60,50]                   Display using F format with 50
digits.
input file.dat                   Inputs commands from file file.dat.
output file.dat                  Outputs user vars and funs to
file.dat.
help polyroot                    Displays a brief explanation of
polyroot.
functions                        Displays a list of all defined
functions.
variables                        Displays a list of all defined
variables.
prompt                           Displays this message.
exit                             Exits this program.
```

Expressions are case insensitive and may be continued on next line by typing \ at end of line.

C. Additional Comments

1. By default, the initial working precision when mathtool is invoked is 100 decimal digits. This can be increased to as high as 1000 digits by using the "digits" command as shown above. If even higher

precision is needed, see the instructions at the start of the globdata.f file. Changing the value of digits does not change the number of digits displayed in interactive output -- by default only the first 68 digits are displayed. To display more (or fewer), see item 7 below. By default, epsilon is set to -digits, meaning a tolerance of 10^epsilon, but it can be adjusted as shown above.

2. User-defined variable and function names are limited to 16 characters (alphabetic, digits, underscore), case insensitive.

3. All constants and variables are stored as high-precision numerical values, not in a symbolic form as with programs such as Maple or Mathematica. Thus if you increase the value of digits (the working precision), you must recalculate all variables that you have defined, unless they were originally calculated to at least the specified precision.

4. The following mathematical constants are pre-defined: e, log2, log10, pi, catalan (Catalan's constant), eulergamma (Euler's constant) and infinity.

5. The following mathematical functions are pre-defined: abs, arccos, arcsin, arctan, arctan2, bessel, besselexp, binomial, cos, erf, erfc, exp, factorial, gamma, integrate, log, max, min, polyroot, pslq, sin, sqrt, sum, table, tan, zeta, zetap, zetaz. Arctan2 is a two-argument form of arctan, similar to atan2 in Fortran; bessel[t] is the same as BesselI[0,t] in Mathematica; besselexp[t] = bessel[t]/exp[t]; gamma is the gamma function; zetap and zetaz are multi-zeta functions -- see http://www.cecm.sfu.ca/projects/ezface/definition.html.

6. If you mistype a function definition, you cannot simply retype it (a syntax error will result). First remove the function name using the "clear" command as shown above. The function definition for an existing function can be displayed by simply typing the function name.

7. For faster execution when computing integrals or sums, pre-define any constant expressions that appear in a function definition. For example, rather than defining f[x] = sqrt[2] * exp[x], first define sqrt2 = sqrt[2], then define f[x] = sqrt2 * exp[x].

8. The format for displaying subsequent interactive results can be changed by using the "eformat" and "fformat" commands, as shown above. These are similar to the Fortran E format and F format, respectively: the first argument is the total length, and the second argument is the number of siginificant digits. The default format is eformat[78,68]. When the first argument exceeds 80, output values are continued on additional lines as needed. The second argument may not exceed the value of digits (the current working precision). For example, if one types "eformat[15,10]", then "pi", the result is ' 3.1415926535e0'. If

one types "fformat[15,10]", then "pi", the result is '
3.1415926535'.

9. Several of the facilities within the mathtool program, notably the quadrature and PSLQ facilities, produce some debug output as their calculations proceed. This output can be controlled by setting debug, which by default is set to 2. All such output (including for instance the CPU time used in each command) can be canceled by typing "debug = 0".

10. The "output" command can be used to save variable and function definitions, as shown above. Variables are saved to an accuracy of the current working precision, as specified by using the "digits" command. The "efformat" and "fformat" commands do not affect the format or accuracy of data written to a file by the "output" command.

11. A facility is available for summing infinite series (or finite sums, for that matter), as shown above. Be advised, however, that the summand function is evaluated directly as shown when computing such sums. This is not a problem for summand functions such as "(n+3)/16^n", which converge rapidly to zero, but summand functions such as "1/(3*n+2)^2" do not converge to zero rapidly enough for the infinite series to be evaluated to high accuracy in reasonable time. In many cases, such sums can be evaluated Maple or Mathematica, which employ advanced transformations.

12. A facility is available for finding roots of polynomials, either real roots or complex roots -- see examples above. The starting value should be reasonably close to the desired root if possible. If you do not have any idea where the root(s) may be, try "random" starting values. Note especially that for finding a complex root, do NOT specify zero as the imaginary part of the starting value, since the resulting Newton iterations will then be restricted to the real line, and the root will not be found.

13. A powerful integration (numerical quadrature) facility is available in the mathtool program. The integrand function may have a blow-up singularity at either or both endpoints of the interval, and either or both of the endpoints may be "infinity" or "-infinity". However, the function should not have a singularity within the interval -- if it does, split the integral into two integrals. Three quadrature schemes are available, namely tanh-sinh, error function and Gaussian quadrature, although by default only the data for tanh-sinh is pre-computed by the mathinit program. To enable all three schemes, change the parameter "nquadopt" in mathinit.f to 3, then recompile mathinit (by typing "make") and re-run. This requires about 2 hours run time on a 2004-era system. Gaussian quadrature is the fastest of the three schemes for continuous, well-behaved integrand functions, but does poorly for integrands with singularities. Error function (erf) quadrature works well even if the function has a blow-up

singularity at one or both of the endpoints of the interval. Tanh-sinh is even faster for many functions. To switch between these schemes, type "quadtype = 1" for Gaussian, "quadtype = 2" for erf, and "quadtype = 3" for tanh-sinh. Each of these schemes performs calculations with multiple "levels" (sets) of abscissas and weights. Each additional level provides additional accuracy, but also approximately doubles the run time. The highest level to be used is controlled by the quadlevel command. By default, 10 levels of abscissas and weights are pre-calculated by the mathinit program, so quadlevel can be set as high as 10. The quadrature routines attempt to find a value accurate to within a tolerance of 10^epsilon. Epsilon can be adjusted as shown above. The quadrature routines, particularly the erf and tanh-sinh schemes, work best when a function with a blow-up singularity at one endpoint is transformed, via a linear transformation, to one where the singularity is at zero. Example: the function f[t] = t/sqrt[1-t^2] has a singularity at 1. So the integral of f[t] from 0 to 1 is better changed to the integral of g[u] from 0 to 1, where g[u] = (1-u)/sqrt[u*(2-u)], and u = 1-t.

14. A powerful integer relation detection facility, employing Ferguson's PSLQ algorithm, is available in the mathtool program. Given n real numbers x_i, an integer relation program finds integers a_i, not all zero, such that a_1 x_1 + a_2 x_2 + ... + a_n x_n = 0. If the largest integer among the a_i has d digits, then each entry of the input real vector (a_i) must be calculated to at least n*d digits accuracy, and the algorithm must be performed using at least n*d digit accuracy, or else any recovered "relation" will not be numerically meaningful. By default, the standard one-level PSLQ algorithm is used, which is adequate for modest-sized problems. If the run time is excessive, a two-level PSLQ program, which performs most PSLQ iterations using double precision (updating the multiprecision arrays only when necessary), is available by typing "pslqlevel = 2". A three-level PSLQ will be available by typing "pslqlevel = 3", although this has not yet been implemented in the mathtool program. By default, PSLQ iterations are abandoned when the norm bound (the maximum Euclidean norm of any possible integer relation vector) exceeds 10^100. This termination condition can be changed by typing, for example, "pslqbound = 40", which terminates when the PSLQ bound exceeds 10^40. A third relevant parameter is epsilon. By default, epsilon is set to -digits, but it can be manually adjusted. Typing "epsilon = -80", for example, means that PSLQ will return a solution vector if its inner product with the input real vector is less than 10^(-80).

V. Note on x86-Based Processors (MOST systems in use today)

The algorithms in this library assume IEEE double precision floating point arithmetic. Since Intel processors have extended (80-bit) floating point registers, the round-to-double flag must be enabled in

the control word of the FPU for this library to function properly under these processors.

The preprocessor macro X86 (defined in config.h, if necessary) will fill in the following functions with appropriate code to facilitate manipulation of this flag. If the flag is not set, the functions do nothing (but still exist). Note that this should automatically be handled by the configure script.

fpu_fix_start This turns on the round-to-double bit in the control word.
fpu_fix_end This restores the control flag.

By default, fpu_fix_start will be called when mp::mp_init is called, so in most cases no user intervention is necessary. However, if this change of rounding mode causes problems with the non-ARPREC portion of the computation, the rounding mode can be explicitly turned on and off by calling fpu_fix_start and fpu_fix_end. For example,

```
int main() {
  unsigned int old_cw;
  fpu_fix_start(&old_cw);
  mp::mp_init( ... )

  ... user code using ARPREC ...

  fpu_fix_end(&old_cw);

  ... user code not using ARPREC ...
}
```

A Fortran-90 example is the following:

```
program foo
use mpmodule
implicit none
integer*4 old_cw

call f_fpu_fix_start(old_cw)
call mpinit( ... )

  ... user code using ARPREC ...

call f_fpu_fix_end(old_cw)

  ... user code not using ARPREC ...

end program
```